Fault-tolerant software in real-time single-chip microcontroller systems

N. Q. BURNHAM and C. F. COWLING

Microcontrollers store their programs in on-chip ROM, and because of the tight linking of the program store and the CPU, it is generally assumed that the program is secure and unaffected by the operational environment. Furthermore, the use of microcontrollers normally results in a greatly reduced component count and often eliminates the need for mechanical components which are subject to wear. Much greater reliability can therefore be expected from equipment using microcontrollers, and in general such high reliability is realized.

However, microcontrollers are now moving into areas where only the very lowest fault levels can be tolerated, and where the consequences of system malfunction can be costly and even dangerous. Applications such as the control of large mechanical or electrical systems, vehicle control, and telephone exchanges are typical examples.

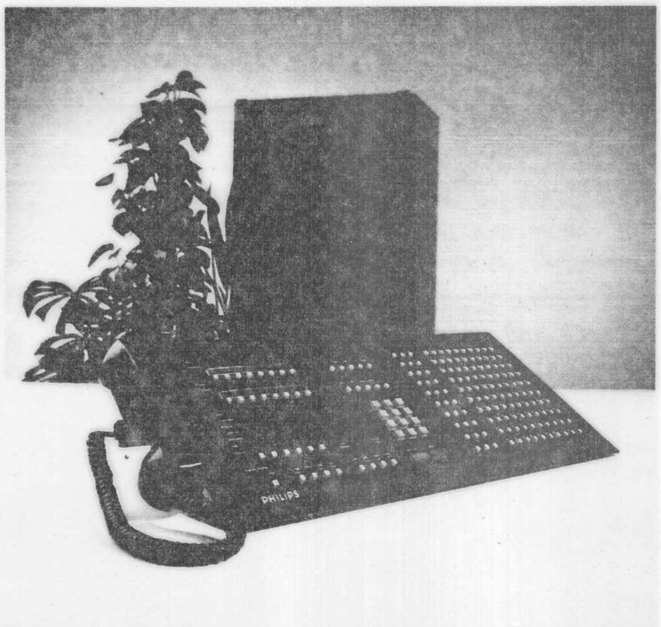
In such applications, secure, uncorrupted and bug-free programs cannot be assumed, and the designer must take measures to ensure that transient program or data errors, or deeply hidden software bugs, do not result in system failure.

This article discusses some of the techniques that can be advantageously used in designing highly fault-tolerant software for microcontroller-based systems. As an example, the design of software for a sophisticated PABX telephone exchange junction using MAB8400 series 8-bit microcontrollers is described. Tests on the system under extreme noise conditions showed that with conventional software, significant unrecoverable errors occurred, whereas the same system operating with the fault-tolerant software proved extremely rugged.

The general techniques outlined are applicable to any microcontroller-based system and should be considered for any system where software failure might prove hazardous.

GENERAL CONSIDERATIONS

Microcontroller-based systems are fragile because for correct performance they rely on a software 'edifice' whose structure can only be maintained by the supply of valid machine code instructions to the CPU. The corruption of just one byte can result in incorrect interpretation, leading to a temporary or even permanent collapse into random logic execution, and the destruction of RAM data. The effect on the system of such a collapse is unpredictable and its severity depends on the application.


The design of fault-tolerant software for interfacing this TBX3000 digital PABX with a public exchange junction illustrates the principles discussed here

As an illustration of potential software fragility, consider a microcontroller-based system executing some 10^5 bytes of machine code per second in normal operation. To perform without failure over a five year period of continuous operation, say, the system must successfully execute about $1{,}6 \times 10^{14}$ sequential bytes. It seems unreasonable to assume that such a long string of machine code operations could be performed without failure, particularly in view of error sources such as noise, program faults, memory faults, or marginal faults in the CPU and associated circuitry.

The difficulties in correcting for such errors are enhanced by the use of variable byte-length instructions, and by the lack of memory error correction. Problems also arise because the microcontroller cannot discern the significance of the machine code being executed (that is, data or instruction code). Therefore, at the most important and fundamental level, there is no mechanism for failure detection. This situation may be alleviated to a significant extent by adopting the design techniques outlined later in this article. However, before discussing methods of achieving fault tolerance, it is worth considering error sources in more detail to obtain an appreciation of their contribution to possible system failure.

Error sources

Program faults

It is generally accepted that bug-free software is a difficult, if not impossible, target. It must be assumed, then, that any software above a trivial level of complexity has resident bugs. Moreover, it is usually impossible in practical and economic terms to subject a system to the full combinatorial range of stimuli in a laboratory environment. Even after field trials, there are usually some faults of a subtle nature left unexposed.

Noise

Noise transients, produced for example by lightning, electromagnetic components in a control system, and heavy duty equipment on the same mains circuit, tend to be underrated as error sources. They give rise to 'apparent software failure' and their effects include:

- corruption of memory Reads, resulting in changed data or incorrect instruction code
- corruption of memory Writes, giving incorrect data at one or more RAM addresses
- initiation of hardware-driven CPU functions such as reset, false-interrupt generation, or processor halt
- interruption of the clock supply; this is rare but possible and will cause indeterminate effects in dynamic processors.

Most of these effects can lead to serious software failure and, because of their transient nature, are difficult to identify. In all systems, it is important that well-proven procedures for circuit decoupling are adhered to.

Memory faults

Strictly, these are hardware failures, and they result in data or instruction code corruption. If they occur in little-used routines, they may give transient-like errors.

Effects of transient errors

In a hardware system, transient errors are unlikely to cause permanent catastrophic failure since the physical integrity of the system (that is, interconnections, chip functions, etc.) is unaltered. This is not the case in a software system where, by analogy, the interconnections and circuit functions are represented by sequences of machine-code instructions and data which, even if embedded in ROM, may be interpreted by the CPU as meaningless (from a systêm function point-of-view) strings of code. It is as if, in a hardware system, the interconnections and circuit functions were being randomly and repetitively re-arranged. Transient errors can result in:

- random logic execution of indefinite but potentially infinite duration
- looping without return to normal sequence
- initiation of halt conditions.

The effect on external devices being controlled by such random processing is clearly unpredictable.

SOFTWARE FAULT TOLERANCE

Implementing hardware functions in software significantly reduces the incidence of trivial system failure. However, the probability of catastrophic failure may increase and the effect on the system can be more severe. Nonetheless, the very real advantages of software-based microcontroller systems (greater flexibility, smaller physical volume, broader application, and lower cost) still outweigh the penalties incurred (more memeory required, less available processing time) provided the fault tolerance of the software is maximized. A reasonable minimum objective for fault tolerance in a software-based system can be stated as follows: *The response of the system to transient failure should be no worse than that of the hardware system it replaces, even though performance and flexibility are improved.*

Improvement in fault tolerance and reliability implies a degree of redundancy and extra processing, resulting in greater memory usage and lower processor availability for prime tasks. This, however, would seem to be contrary to the aims of most system designers, who would reasonably be seeking minimum memory utilization and short program execution times. However, reliability has to be paid for, and experience suggests that an overhead in program code for diagnostic routines of 5% to 15% will give a significant improvement (except where exceptional security is required). The additional code that this implies is, of course, system dependent.

Because the CPU cannot differentiate between valid and invalid strings of code, it is not possible to prevent misoperation occurring. It is possible, however, to detect misoperation once it has occurred. Fault tolerance ultimately depends on the effectiveness of the diagnostic and recovery procedures employed. The following points are recommended for serious consideration.

- Consider the diagnostic check and recovery procedures as an inherent part of the design; do not add recovery mechanisms as an afterthought.
- Determine the effect of potential failure in the operational environment; this defines the degree of security and the sophistication of the recovery procedures required. It also determines whether duplication of software is necessary.
- Aim for simplicity in software, data, and supervisory structures. In practice, complex software requires complex recovery mechanisms.
- Ensure that there is at least one restart address to which software execution can be forced by external means. This guarantees escape from random logic execution.
- Minimize the sub-routine nesting depth to simplify the recovery methods. If possible, keep to a single level (that is, for interrupt handling). Accept the resultant increase in program code as a reasonable trade-off for ease of recovery.
- Provide back-up in RAM for that data which is essential for software supervision. This will permit 'intelligent' restart.
- Fill every vacant memory location with single-byte instructions and provide a jump to a 'safe' destination at the end of each infill area. This 'traps' invalid jumps into vacant memory since it provides a secure exit mechanism.
- Determine the smallest real-time period within which recovery must be achieved. This defines the required frequency for diagnostic checks and the response time for recovery.
- Examine the effects of an invalid reset (caused, for instance, by noise). For example, if power-on reset initialization would be catastrophic should it occur at any other time, then the reset invoked during recovery must omit the initialization procedure. In this case, a latch can be used to determine that power-on reset has already occurred.
- Keep in mind the technology and architecture of the processor to be used. For example, compared with dynamic processors, static types are less affected by an interruption in the clock supply. Note also that processors operating with three-byte instructions are more likely to enter into random logic execution than two-byte instruction devices. A benchmark comparison indicates that systems using two-byte instruction devices are two to three time more secure against crashes due to program fetch errors than those using three-byte types.

Error prevention

In the initial design phase, action should be taken aimed at preventing incorrect operation by, for example, 'tying down' unused interrupt vectors and 'padding out' branch tables with defined exits. For example, interrupt vector locations should always be filled so that if incorrect arrival at the vector occurs, the program will be directed to a safe destination. Similarly, branch tables not completely filled should be padded out with default values to safe destinations.

Maintenance of integrity

It is essential that vital control variables such as RAM pointers and table indices are kept within valid bounds. (For example, index registers used with branch tables of length, say, 16, should be checked for values not greater than 16.) Such containment can, under fault conditions, prevent an excursion into uncontrolled random logic execution although it will not necessarily prevent some illogical function sequences.

Data preservation

Variable data is repetitively set up, in contrast to a once-only operation, thus providing a continuous data recovery facility without the need for specific administration and diagnostic procedures. For example, an instruction issued to a critical I/O device cannot be assumed to be indefinitely valid; it should be reinforced periodically.

FAULT-TOLERANT SOFTWARE IN A PRACTICAL APPLICATION

As an example, the design of fault-tolerant software for a U.K. public exchange junction interfacing with a TBX3000 digital PABX is described (Fig.1). To keep component count and cost to a minimum, the system is implemented using MAB8400 microcontrollers with external EPROMs, each time-shared over four circuits (Fig.2 and 3). Maximum use is made of software in realizing circuit functions such as the detection of telephone ringing waveforms and contact debounce. Although the design is inexpensive and reliable, it is vital that the system runs for long periods without catastrophic failure. Malfunction on just one junction card can result in the loss of four external lines.

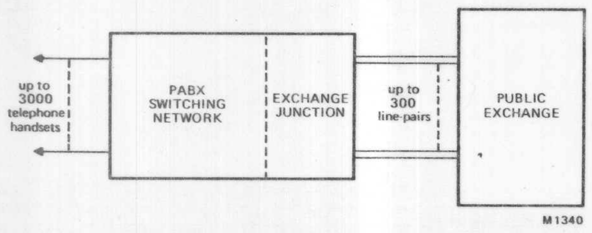


Fig.1 Public exchange/PABX interface

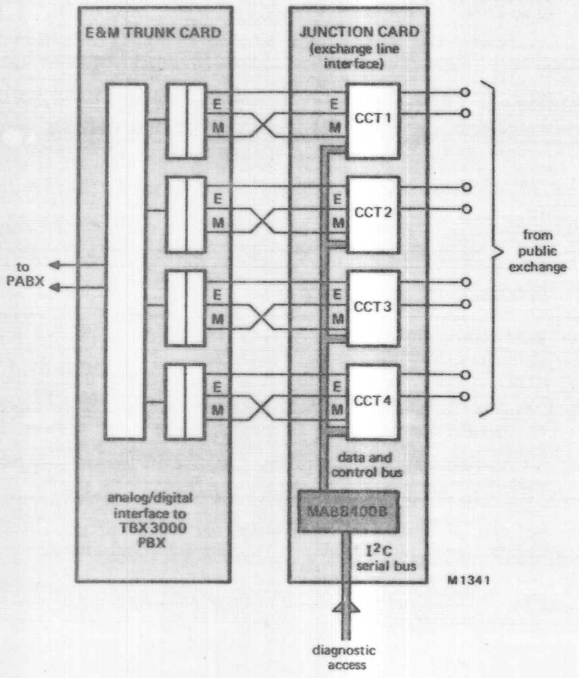


Fig.2 Public exchange/PABX junction interface using an MAB8400 to control four circuits

The traffic handled on a PABX/public-exchange junction is usually heavy in the 'busy' hour. The loss of even a small number of circuits during this period, with the consequent increase in traffic on the remainder, can lead to an objectionable increase in route blocking, particularly in small systems. Even a trivial processing error, if not recovered, can have a disproportionate effect on system operation since it is likely to lead to looping or random logic execution. Two serious effects of this are:

- Permanent 'lock-out' of the four circuits. This need not necessarily render the circuits busy to new calls; the circuits may accept calls but not switch. This can be particularly aggravating in certain types of business (e.g. mail order).
- Generation of spurious line signals. This leads to intermittent seizure of the public exchange and false indication of incoming calls at the PABX.

In the system described, the MAB8400 junction processor has no communication with the TBX3000 CPU and is therefore totally reliant on autonomous measures for recovery in the event of a fault condition. Furthermore, the junction design does not include a hardware watchdog timer, and therefore recovery must be achieved entirely by software means.

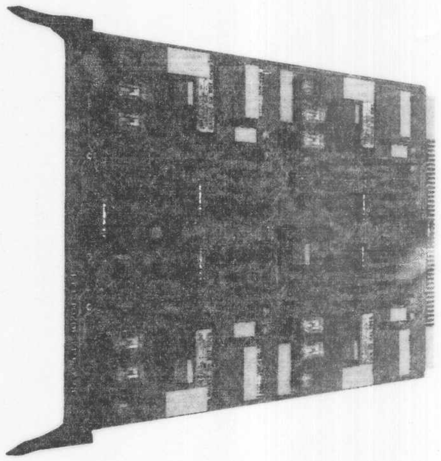


Fig.3 Junction card

Software structure

The software is partitioned into two distinct sections: Real-Time Software (RTS) and State Processing Software (SPS) (see Fig.4).

The Real-Time Software is interrupt-driven from the internal timer at 5,5 ms intervals and controls all I/O functions, input signal integration, digit regeneration, impulsing (10 impulses per second), ringing waveform analysis, circuit selection and scanning, and timer interval updating.

The State Processing Software controls the telephony functions on a state-driven basis. State transitions are handled by a supervisory program (part of the SPS).

A detailed diagram of the software structure is shown in Fig.5.

Reliability measures

The term 'catastrophic failure' in this application does not imply any hazard. An occasional call failure can be tolerated, and consequently duplication in software or special measures to guard the reset condition are not necessary. The real-time telephony environment will allow a slip of two 5,5 ms system real-time periods for diagnostic and recovery action before any significant degradation of performance occurs. Recovery requires restart of internal timer, initialization of the sub-routine stack level, and restart of the State Processing Software to continue execution of the current system state.

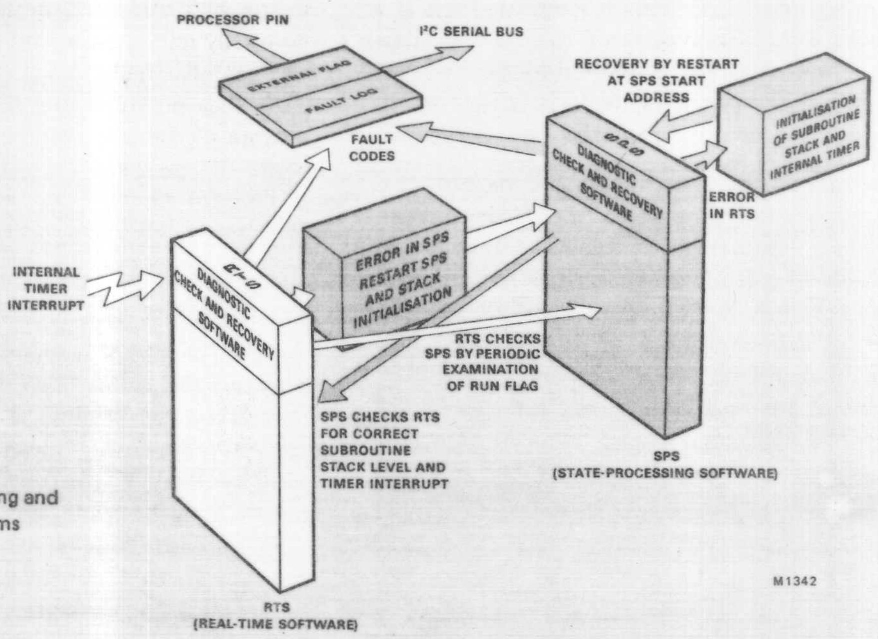

PROCESSOR PIN
I²C SERIAL BUS
EXTERNAL FLAG
FAULT LOG
FAULT CODES
RECOVERY BY RESTART AT SPS START ADDRESS
INITIALISATION OF SUBROUTINE STACK AND INTERNAL TIMER
DIAGNOSTIC CHECK AND RECOVERY SOFTWARE
ERROR IN RTS
INTERNAL TIMER INTERRUPT
ERROR IN SPS RESTART SPS AND STACK INITIALISATION
RTS CHECKS SPS BY PERIODIC EXAMINATION OF RUN FLAG
SPS CHECKS RTS FOR CORRECT SUBROUTINE STACK LEVEL AND TIMER INTERRUPT
SPS (STATE-PROCESSING SOFTWARE)
RTS (REAL-TIME SOFTWARE)
M1342

Fig.4 Software partitioning and dignostic check mechanisms

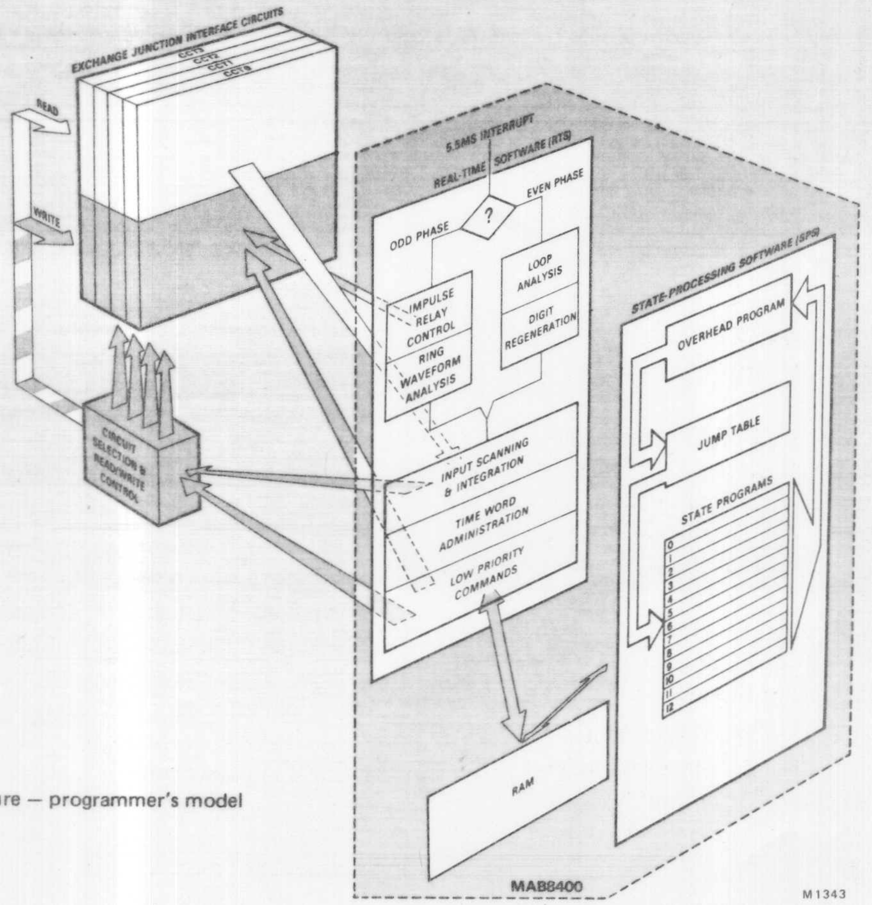


Fig.5 Software structure – programmer's model

RAM data is not backed up in this application since dependence on historical RAM data is minimal. There is a strong likelihood that, in the event of RAM-data corruption, the system will reprocess the current telephony input signals and arrive at the correct state, or at a state which will maintain transmission between the two parties involved in a call.

Microcontroller sub-routine level

Since the sub-routine stack is open to corruption, its level is limited to one; that is, there is only one sub-routine, the complete RTS. The stack level is zero during state processing. Since there are no nested sub-routines, the recovery procedure is greatly simplified, albeit at the expense of some additional PROM.

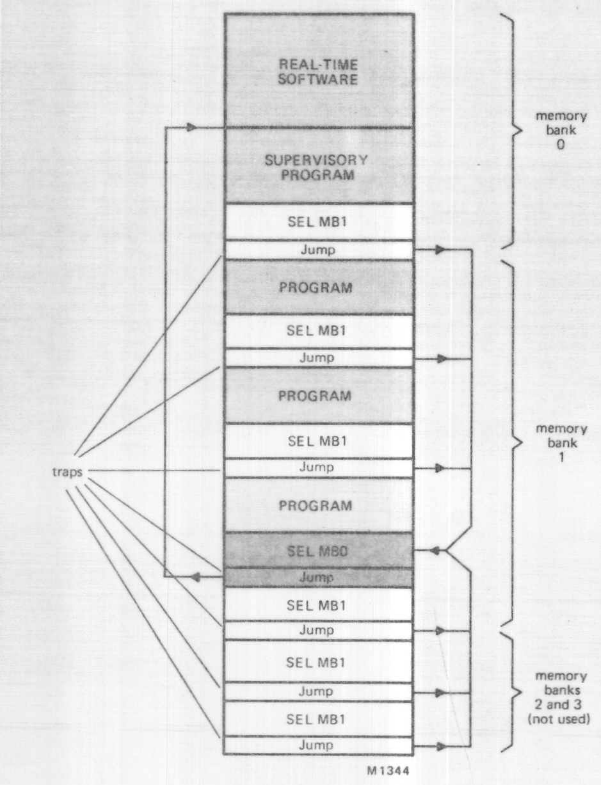


Fig.6 Vacant memory infill and traps

Traps

In this system, a timer/counter (T/C) interrupt is the only valid interrupt. The effects of false external or serial I/O interrupts are limited by placing RETR (return from interrupt sub-routine) instructions at the appropriate vector locations.

A T/C interrupt, invoking the RTS, should only occur during SPS execution. Should the interrupt occur during RTS execution, it is a fault and will be recognized as such when the RUN flag in the state program is checked. If the RUN flag is invalid, the RTS returns to the start of the supervisory program by 'forcing' the start address in the return address stack.

Program modules are mapped for linking, with gaps deliberately left between them so as to avoid the possibility of overrunning page and memory boundaries. One result of this is that sections of memory between programs, and between the last program and the end of memory, are vacant. This vacant memory is used in the system to 'trap' improper jumps; see Fig.6. An unconditional jump instruction is inserted prior to each program and at the end of memory. Remaining vacant areas are filled with 'Select Memory Bank 1' (SEL MB1) instructions. Should an improper jump be made into an unused memory area, then the jump instructions return control to the supervisory program located in Memory Bank 0, via intermediate SEL MB0 and an unconditional jump instruction.

The vacant memory infill bytes represent *single-byte* instructions and therefore guarantee sequential program execution until the next trap jump is required.

Note that the SEL MB instructions preselect the required bank and are only effective when an unconditional jump is executed.

OPERATIONAL RESULTS

The vulnerability of the system to fault conditons was observed under laboratory conditions both with and without the recovery mechanism in operation. With the prototype junction card installed in a TBX3000 laboratory system, and without the recovery mechanism in operation, occasional catastrophic failures were observed. These were significant, particularly in view of the electrically-benign conditions under which they occurred.

A large increase in the rate of catastrohpic failure occurred when the junction card was used to verify the design of a production junction card tester. The increased failure rate was found to be due to noise, produced by 'unquenched' reed relays, being transmitted back via a +12 V power supply into the mains, and from there via a +5 V supply to the MAB8400 supply pins.

With the recovery mechanism again not operating, the system was subjected to 30 ms, 3 V peak-to-peak noise bursts at the rate of 10 bursts/second, applied to the +5 V power rail. The noise was generated by a T51 relay operating via a self-interrupting contact; the noise spectrum was similar to that of the production tester. Failure modes similar to those observed previously occurred and analysis showed them to be either random logic execution, internal timer disablement, or reset. (The occurrence of reset was relatively rare compared to the other fault conditions.)

Under the above conditions but with the recovery mechanism now operating, no catastrophic failures were observed. Subsequent monitoring of fault recovery flags indicated that failures were indeed occurring but that the recovery mechanism was entirely effective.

It was concluded from these results that the degree of fault tolerance designed into the system is sufficient to meet operational field requirements. This conclusion was reinforced by further trials in which software execution was started at 'random' memory addresses (even in the middle of multi-byte instructions) in an attempt to induce catastrophic failure; no failures were observed. (The latter tests were carried out using a microcontroller development system in debug/ICE mode.)

FURTHER MEASURES

In some systems, further integrity may be desirable; the techniques described above can be enhanced as follows.

System logic breakdown

The software described here is based on the state transition method, and it is possible to validate state changes by reference to a table of permissible state transitions (current and previous transitions are recorded). Depending on the required design sophistication, system logic breakdown can be handled either by exiting to specific fault handling states, or by recursion to the previous state. In applications where the response of the external environments is slow compared with the recovery time (as is so in telephony), recursive handling is obviously the simplest procedure to adopt.

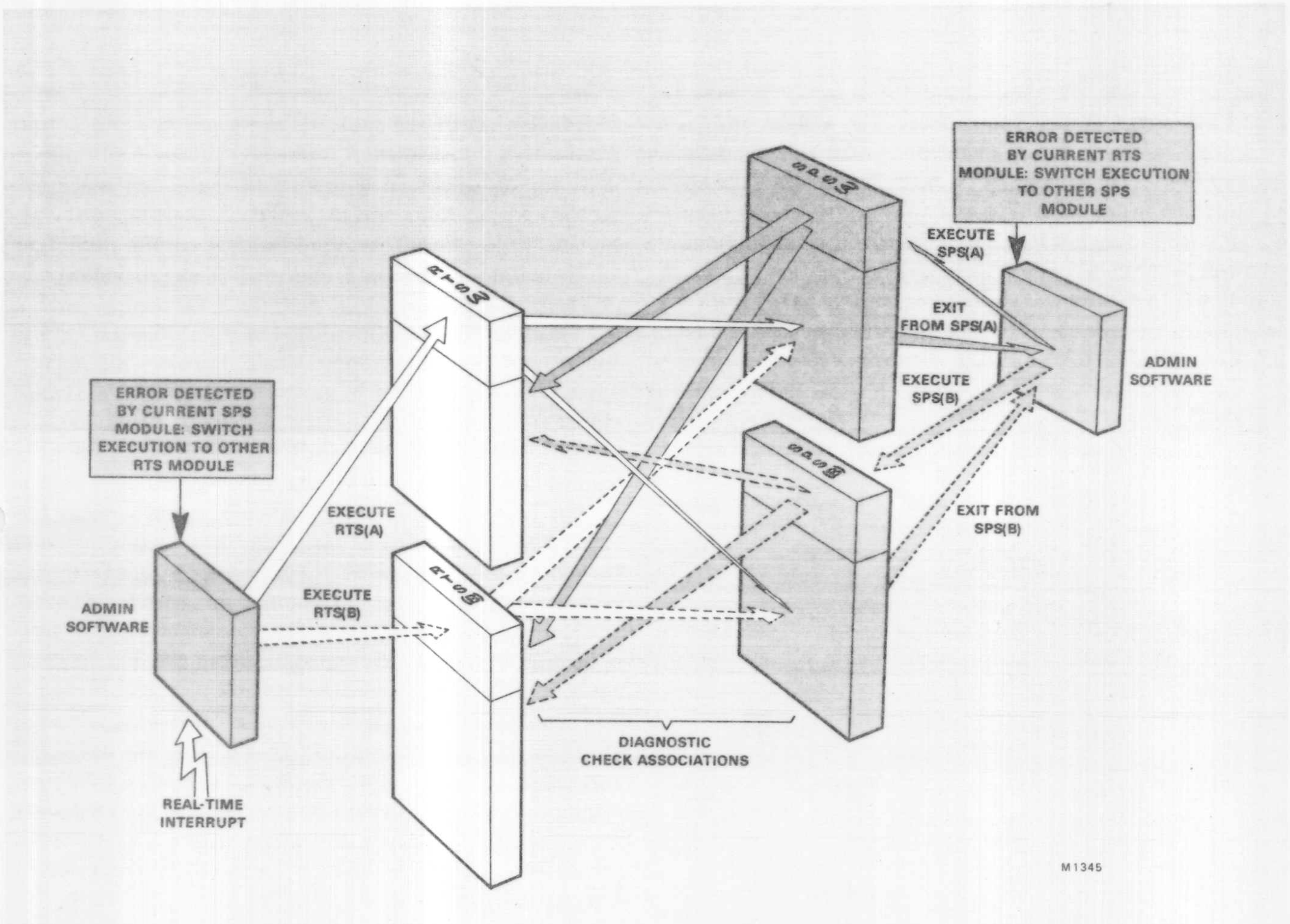

ERROR DETECTED BY CURRENT RTS MODULE: SWITCH EXECUTION TO OTHER SPS MODULE
EXECUTE SPS(A)
EXIT FROM SPS(A)
EXECUTE SPS(B)
ADMIN SOFTWARE
EXIT FROM SPS(B)
ERROR DETECTED BY CURRENT SPS MODULE: SWITCH EXECUTION TO OTHER RTS MODULE
EXECUTE RTS(A)
EXECUTE RTS(B)
ADMIN SOFTWARE
REAL-TIME INTERRUPT
DIAGNOSTIC CHECK ASSOCIATIONS
M1345

Fig.7 Basic configuration for duplicated software

Memory failure

In some microcontrollers, extensive memory expansion is possible. Although failures in this memory are strictly hardware failures, in operational terms, the effects are identical to those caused by other factors; that is, the outcome is an apparent software failure with all the potential effects outlined earlier.

The normal method of overcoming this problem is to perform periodic tests on RAM and checks on the validity of program memory, further processing being aborted under fault conditions. Whilst this is a fail-safe procedure, it is not fault tolerant.

To achieve fault tolerance in program memory, redundancy must be incorporated in the system either in the form of error correction or direct duplication of program. For the design described here, the latter is the only option. Thus both real-time and state-processing software modules are duplicated (see Fig.7).

Any of the four operating combinations are permissible. During recovery, RTS/SPS combinations are reassigned. If, for instance, a failure is detected by an RTS module in an SPS module, the restart is switched to the supervisory program of the second SPS module. The procedure is identical for a failure detected during SPS execution.

At the cost of a small increase in diagnostic intelligence, discrimination can be made between permanent failure and transient failure. This leads to the interesting possibility that, in the event of recording a permanent failure in both modules in one half of the system, the failure is likely to be due to pure software error since permanent memory faults are unlikely to occur in both modules simultaneously.